7800 PRO-SYSTEM KEYBOARD SOFTWARE GUIDE

Version 1.1 - June 19, 1984

INTRODUCTION

The 7800 Pro-System Keyboard is a computer keyboard that plugs into the right hand (player 2) joystick port of the 7800 Pro-System, transforming it into a low-end home computer. The computer uses the same peripheral as the Atari home computer line. This guide is intended to provide the necessary information for software developers to use the keyboard.

This is the first version of the software guide. As such, it will probably contain several errors in it. If you notice such an error, please contact me ASAP so that I can correct it in future versions. Also, this is a software guide, not a hardware specification.

KEYBOARD CAPABILITIES

The 7800 Keyboard is capable of performing two main functions. One, it functions as a full stroke keyboard with 2-key rollover. Two, it provides access to I/O devices such as printers, disks, the ATARI 1010 Program Recorder and cassette recorders.

The keyboard is a 63 key full stroke keyboard. The layout is identical to the layout of the keyboard on an ATARI 800 Home Computer. The five function keys on the right of the keyboard have different labels. The top is labelled help, and the next 4 have different geometric shapes as labels. The keyboard will provide 2-key rollover. That is, if you were to hit A and B at the same time, then depress C also, then release A, and then release B, the result would be the key presses 'ABC'. This may seem a bit strange at first, but this sort of sequence happens quite often with touch typists. The keyboard supports repeat keys for all keys. It should be noted that holding 3 keys down at once defeats the 2-key rollover; no keys will be sent until 1 is released. This does not apply to the CONTROL or SHIFT keys. Use of CONTROL and SHIFT at the same time will not always work (as on the Atari 800).

The keyboard has an ATARI Asynchronous Serial Input/Output (SIO) bus connector in the back. Through it, access to existing ATARI I/O devices is provided. All time critical functions are handled by the keyboard. The following devices can be supported through this connector: disk drives, printers, the 1010 Program Recorder, and the 850 interface module.

The keyboard also has two audio jacks to enable it to connect to ordinary cassette recorders. The keyboard supports read and write record operations. The keyboard uses the same record format for cassettes as used by the ATARI 800 OS for the program recorder. The data format used is similar to that used by the Apple II for tape storage.

GETTING STARTED

To understand how to program the keyboard it is necessary to understand the communication protocols used to talk to the keyboard. (NB this is not true if you want to use the standard communication routines currently available in the FIC directory and discussed in section 8.) First a diagram of the interface.

```
+----------------+                                            +---------------
+|               |                 BCLK                       |
|
|                |------------------------------------------->|
|                |                 KCLK                       |
|                |<-------------------------------------------|
|                |                  D2                        |
|   BASE UNIT    |<------------------------------------------>|     KEYBOARD
|                |                  D1                        |
|                |<------------------------------------------>|
|                |                  D0                        |
|                | ------------------------------------------>|
|                |                                            |
+----------------+                                            +---------------
```

Figure 1. Hardware interface between 7800 Pro-System and keyboard

If the keyboard is plugged into the right hand joystick port (as is standard), the KCLK line is read as the MSB of INPT5 (just as if it was the fire button on a joystick), the BCLK line is bit 3 of SWCHA, and D2-D0 are the 3 LSB's of SWCHA. Lines D2-D0 are bi-directional. The direction is under the control of the base unit though CTLSWA.

Clock Lines:

There are two clock lines: BCLK and KCLK. BCLK is bit 3 of SWCHA. It is under the control of the Base unit, hence BCLK. It should be configured as output always. This requires bit 3 of CTLSWA to be set to 1. KCLK is the keyboard's clock line. It is read as the MSB (bit 7) of INPT5. This is the same as if it was a fire button on a joystick.

3 Bit Communication Protocol

All communications are initiated by the base unit (the software running on the 7800 Pro-System). To initiate communication, the base unit toggles BCLK. To respond, the keyboard will toggle KCLK. All 3-bit communications require 1 and only 1 toggle from each component (the base unit and the keyboard).

To send 3 bits of data to the keyboard:

0) The base unit configures D2,D1,D0 as output. (Set 3 LSB's of CTLSWA to 1's.)

1). The base unit puts valid data on D2,D1,D0 and toggles BCLK. (This may be done simultaneously.)

2). Data must be kept valid until the keyboard toggles KCLK.

To receive 3 bits of data from the keyboard:

0) The base unit configures D2,D1,D0 as input.

(Set 3 LSB's of CTLSWA to 0's.)

1). The base unit toggles BCLK. (Request for data)

2). The keyboard toggles KCLK when data is valid. (Data will stay valid until next BCLK toggle)

Sending a byte at a time:

To send or receive 1 byte of data requires 3, 3 bit communications. The data is sent least significant 3 bits first (then middle 3 then upper 2). Thus, to send a byte whose value is b7|b6|b5|b4|b3|b2|b1|b0 the base unit would:

1) Send 3 bits with D2 = b2, D1 = b1 and D0 = b0.

2) Send 3 bits with D2 = b5, D1 = b4 and D0 = b3.

3) Send 3 bits with D2 = X (don't care), D1 = b7 and D0 = b6.

To read a byte with value b7|b6|b5|b4|b3|b2|b1|b0

1) Read 3 bits. D2 = b2, D1 = b1, and D0 = b0.

2) Read 3 bits. D2 = b5, D1 = b4, and D0 = b3.

3) Read 3 bits. D2 = X, D1 = b7, and D0 = b6.

Keyboard Commands:

The keyboard is set up as a slave peripheral device. That is, it will always perform functions in response to commands from the base unit. For example, if the base unit wishes the keyboard to function as a keyboard, the base unit sends a SCAN-KEYBOARD command to the keyboard. The keyboard would then commence keyboard scan.

The keyboard is set up to have a nested command structure. The top level commands are SCAN-KEYBOARD, SIO-OPERATION, PROGRAM-RECORDER-OPERATION, CASSETTE-RECORDER-OPERATION, and RETURN. The first four operations will be discussed in detail below. The RETURN operation is an operation present at all levels of the nested command structure. It is defined as returning the keyboard to the command level higher up the tree. For the top level, RETURN merely leaves the keyboard in the top level.

All commands to the keyboard are 3 bits long. Thus it only takes one 3-Bit communication to send a command (1 handshake). The RETURN command is defined to be the value 7 on all levels. It is this definition that is the key to synchronizing with the keyboard.

Synchronizing:

The base unit is 'synchronized' with the keyboard when each agrees what the state of the clock lines are, and what level the keyboard is at in its command structure. The base unit can be sure that the clock line states are in agreement if it toggles BCLK, and the keyboard responds with a toggle of KCLK. To ensure that the keyboard is at a known command level, the base unit should send the keyboard enough RETURN commands so that the keyboard is guaranteed to be in the top level.

To synchronize with the keyboard, assuming the keyboard is at a random point in its command tree:

1) Read the current KCLK state.

2) Do 420 times:

3) Toggle BCLK with D2-D0 = 7.

4) Look for KCLK toggle 128 times. If not found, GOTO step 1
   If found then loop.

Note - The 420 loop value derives as follows. The keyboard could, conceivably, be in the process of sending a [illegible] data sector. If so, 132 data bytes must be sent, plus a return code before the keyboard looks for a new command. This requires 399 handshakes. Then 2 RETURNS to get back to top level (401). The SYNCH procedure uses 420 RETURNS just in case there is a longer path I haven't envisioned (unlikely).

POLLING_THE_KEYBOARD

In order to read key presses from the keyboard, it is necessary to poll the keyboard. I suggest that the keyboard be polled once per frame, as this will allow the keyboard to be properly debounced. Polling 1 time per frame will also provide for straight-forward repeat keys.

To set the keyboard into SCAN-KEYBOARD mode, from the top level, the ba unit should send a 3 bit command with the value zero (0). The keyboard will now be at the SCAN-KEYBOARD level. The valid commands at this level are GET-KEY (0), GET-STATE (1), and RETURN (7). GET-KEY will return the current key press and then return to the SCAN level. GET-STATE will send 3 bits of stat information and then return to the SCAN level. RETURN causes a return to the TOP level.

Getting a key:

When the keyboard is at the SCAN level, and a GET-KEY (0) command is ser to it, it will return a 1 byte value to the base unit that is the current ke pressed. The sequence of operations is:

0) Assume keyboard in top level.

1) Send a SCAN command (0) to keyboard.

2) Send a GET-KEY command (0) to keyboard.

3) Configure for input from keyboard. (3 LSB's of CTLSWA to 0)

4) Read in current keypress. (A read byte operation)

5) (Assuming keyboard polling to continue). Configure for output to keyboard. (3 LSB's of CTLSWA to 1)

6) Goto step 1.

The 1 byte value the base unit receives is not an ASCII value; it needs to be translated. The format of the byte is:

CONTROL|SHIFT|R2|R1|R0|C2|C1|C0

Where R2-0 is the row number, and C2-0 is the column number of the matrix. Control will be 1 if the control key is pressed. Shift will be 1 if the shift key is pressed. Here is the translation matrix for any key pressed assuming that control and shift are 0.

7800 PRO-SYSTEM KEYBOARD TRANSLATION MATRIX

KEYPRESS	KEYCODE	ATASCII	KEYPRESS	KEYCODE	ATASCII
no key	$00	---	'.'	$20	$2E
'6'	$01	$36	'3'	$21	$33
'u'	$02	$75	'x'	$22	$78
'y'	$03	$79	'e'	$23	$65
'7'	$04	$37	'0'	$24	$30
BREAK	$05	---	'd'	$25	$64
'n'	$06	$6E	','	$26	$2C
no key	$07	---	'c'	$27	$63
'j'	$08	$6A	'+'	$28	$2B
HELP	$09	---	'2'	$29	$32
func 2	$0A	---	'-'	$2A	$2D
func 3	$0B	---	'w'	$2B	$77
func 4	$0C	---	'<'	$2C	$3C
'h'	$0D	$68	's'	$2D	$73
' '	$0E	$20	'/'	$2E	$2F
func 5	$0F	---	'/'	$2F	$7B
'k'	$10	$6B	'*'	$30	$2A
'5'	$11	$35	'1'	$31	$31
'i'	$12	$69	'='	$32	$3D
't'	$13	$74	'q'	$33	$71
'8'	$14	$38	[illegible]	$34	$3E
'g'	$15	$67	'a'	$35	$61
'm'	$16	$6D	ATARI	$36	---
'b'	$17	$62	'z'	$37	$7A
'l'	$18	$6C	no key	$38	---
'4'	$19	$34	ESC	$39	$1B
'o'	$1A	$6F	EOL	$3A	$9B
'r'	$1B	$72	TAB	$3B	$7F
'9'	$1C	$39	BSPACE	$3C	$7E
'f'	$1D	$66	CAPS	$3D	---
','	$1E	$2C	no key	$3E	---
'v'	$1F	$76	no key	$3F	---

Special keys

BREAK - The break key has a code of $05. In order to implement the normal break key function, the base unit code must recognize this key. Note: the break key is also accessible through the 3 bit status information (see below).

CAPS - The caps key has a code of $3D. The caps key function is left to the base unit code.

ATARI - The ATARI key (the rectangle that is half white and half black) has code of $36. The use of this key is application dependent. For example, the word processor uses it to underline, and BASIC uses it as a pause key. The ATARI key can be accessed though the 3 bit status information (see below).

HELP - The HELP key has a code of $09. The base unit code is expected to recognize the help key.

FUNCTION keys - The 4 function keys below the help key (2-5 in the matrix above) are labelled with geometric shapes. Their use is application dependent.

REPEAT KEYS

Since the key press returned by the keyboard is always the current key being pressed, the base unit code is responsible for programming repetitive keys. The following algorithm seems to work well.

```
START:
     RptTimer := 0;
      ldKey := 0;

     NewKey := GetKey()    ;  GetKey gets a key press from the keyboard
     If NewKey = 0 Then OldKey := 0; RptTimer := 0; RETURN (0);
     Else
          If RptTimer <> 0 Then Goto DoRepeat
               Else
GotNewKey:
                    OldKey = NewKey; RptTimer := 30; RETURN (NewKey);

DoRepeat:
     If NewKey & $3F <> (OldKey & $3F) Then Goto GotNewKey
     Else RptTimer := RptTimer - 1;
          If RptTimer > 0 Then RETURN (0);
               Else
                    RptTimer = 6;
                    RETURN (NewKey);
```

Notes - This polling routine should be called once a frame (60 times a second). It doesn't do the ATASCII translation. If called once a frame, the delay before a key starts repeating = 30/60 seconds (1/2) and the rate of keystrokes while repeating = 60/6 (10) per second. The comparison on NewKey masks off the Control and Shift bits so that the use of those keys is less prone to errors. (Try typing 'A' and releasing the shift key 1st.) The PIC directory (discussed below) contains an implementation of keyboard polling in the file KEY.S

Keyboard Status Information

Some applications have found it convenient to get status information from the keyboard while not actually reading in a full key. To get status information, send a GET-STATUS (1) command to the keyboard. The keyboard will return a 3 bit status and then return to the SCAN level.

The format of the 3 bits is:

ATARI|ANY|BREAK

BREAK = 1 if BREAK key currently pressed, 0 otherwise.
ANY = 1 if any key is currently pressed, 0 otherwise.
ATARI = 1 if ATARI key is currently pressed, 0 otherwise.

Technical Notes

Here is a little background information on how the keyboard works at the SCAN level. These notes are intended as an aid to debugging.

1) When the keyboard 1st enters SCAN mode, the 1st keypress sent will n code 0.

2) After sending a keypress or a status code, the keyboard goes off and scans the key matrix again, before checking for a new command. This means that if an application tries to send two GetKey's too quickly (right after each other), the 2nd will have to wait until scan ends. Also, sending a GetStatus command will cause a new scan, updating the value of the keypress.

3) The keyboard only scans the keyboard after a GetKey or GetStatus. So, if an application does a GetKey, then waits 4 minutes, and then does another GetKey, the returned code will be 4 minutes old. This is only true i the keyboard is left at the SCAN level for those 4 minutes.

4) The rationale for the keyboard working as described is that a GetKe or GetStatus, if called once per frame, will respond immediately. This is a big time savings for applications which poll in their kernel (most do this).

PRINTER:

DEVICE ID: $40 FOR SIO PRINTERS & THRU 850'S PARALLEL PORT

COMMANDS:

PRINT LINE - $57 AUX1 = PRINT MODE ($4E NORMAL, $53 SIDEWAYS, $44 DOUBLE WIDTH), AUX2 = DON'T CARE WRITE OPERATION 40, 29, 21 BYTES DEPENDING ON PRINT MODE

GET STATUS - $53 AUX1, AUX2 = DON'T CARE READ OPERATION - 4 BYTES.

Common SIO Bugs

Here is a listing of common bugs programming the SIO interfacer as well as suggested solutions. Most of them have been mentioned, in passing, above.

1) SIO operation causes loss of synchronization with keyboard.
   a) Too few or too many data bytes were read/sent.
   b) Return code was not checked.
   c) Error code was read when no error/not read when there was an error.
   d) Forgot to read/send the checksum.
   e) Attempt to use keyboard for other operations during SIO operation. Commonly, polling the keyboard for BREAK.

2) SIO device times out.
   a) Hardware error (check SIO cables, power etc.). If nothing seems wrong then try the suspected device on an Atari Home computer.
   b) Kernel or other interrupt driven code runs too long. Try with DMA off to disable NMI's. Solution, use flag to cut off kernel.
   c) Too much DMA time (never been encountered).
   d) On a write operation, spending too much time between sending data byte sends cause peripheral to time out.
   e) Command frame is incorrect.

3) Peripheral sends NAK in response to command.
   a) Checksum calculated incorrectly.

4) Framing error.
   a) On a read operation, the base unit waits too long between reading in reading in data bytes (commonly due to attempts to process data while reading it in). Recommended procedure is to read data frames into a buffer and then process.
   b) Hardware error.

5) Undefined error.
   a) Please let me know ASAP.

c) Send the 1st auxiliary byte.

d) Send the 2nd auxiliary byte.

e) Send the checksum of the command frame. (Same checksum as SIO)

3) Send the timeout value (in seconds). The keyboard will look for COMPLETE this long before timing out.

4) Send the remainder of the data bytes to be written.

5) Send the checksum of the data bytes.

6) Read in a 3 bit return code. If the code = 0 then done.

7) If code <> 0 then read a 1 byte error code.

Notes:

1) If a write operation does not work, try it with DMA off. If it works then, the base unit code is probably sending the data bytes to slowly. The solution is to short-circuit interrupt driven code during SIO operations.

2) Remember to send all data bytes and the checksum.

3) Remember to only read the error code (1 byte), if the return code is non zero.

SIO DEVICES AND COMMANDS

Here is a short list of SIO devices and command frames. More complete information can be obtained from the SIO User's Handbook.

DISK DRIVES

Device ID's :- $31,$32,$33,$34 (DISK DRIVE #'s 1,2,3,4)

Commands:

GET STATUS - $53 AUX1,AUX2 = DON'T CARE READ COMMAND - 4 BYTES

PUT SECTOR - $50 AUX1 = LSB SECTOR, AUX2 = MSB SECTOR WRITE COMMAND - 128 BYTES

PUT SECTOR (WITH VERIFY) - $57 AUX1 = LSB SECTOR, AUX2 = MSB SECTOR WRITE COMMAND - 128 BYTES.

GET SECTOR - $52 AUX1,AUX2 = DON'T CARE READ COMMAND - 128 BYTES

FORMAT DISK - $21 AUX1,AUX2 = DON'T CARE READ COMMAND - 128 BYTES

3) Send the timeout value (in seconds). The keyboard will look for COMPLETE this long before timing out.

4) Receive the number of bytes expected. Same number as sent in 1.

5) Receive the check sum of these bytes.

6) Receive a 3 bit return code. If code = 0, then done.

7) If code <> 0 then read a 1 byte error code.

Notes:

1) The keyboard will always send the specified bytes of data, even if no peripheral is present. It sends dummy data if it times out.

2) Failure to read in the return code or, where applicable, the error code will result in loss of synchronization.

3) The following error codes are supported: $54 = Timeout; $46 = Framing error; $4E = device sent NoAcknowledge (usually checksum problem)

4) The SIO protocol defines checksums as the one byte sum of all data bytes with the carry added back in each time.

5) Be sure the data lines are configured in the correct direction during each operation. To send data to keyboard, configure for output (3LSB's of CTLSWA (located at $2B1) to 1). To receive data from keyboard, input (3LSB's to 0).

6) When reading data from a disk, it is very important to read in the bytes as quickly as possible. DMA can stay on. However, interrupt driven code, such as the kernel, ought to be short-circuited (i.e. return as close to immediately as is feasible).

7) If a read fails, and the device is there and powered on, try the same read with DMA off. If it then works, the fault is probably with interrupt code. If it doesn't work, the blame is probably your read code.

SIO WRITE OPERATIONS

To get to the SIO WRITE level from the SIO level, send a 1 command. Here is a summary of what the keyboard expects during an SIO write operation.

1) Send the first data byte to be written. This is done so as to precisely time the gap between the CMD frame acknowledge from the peripheral and the start of the data frame.

2) Send the number of bytes to be read.

3) Send the command frame as follows

a) Send the SIO device ID.

b) Send the SIO command byte.

CONFIDENTIAL

SIO_OPERATIONS

The second major capability of the keyboard is support for the Atari Asynchronous Serial Input/Output Bus (aka the SIO bus). The keyboard supports operations that send data frames to peripherals (WRITES) and those which receive data frames from peripherals (READS). The immediate operations, those involving no data (currently supported by no peripherals) are not directly supported. Immediate operations could be programmed as READS of 0 bytes of data.

SIO operations are accomplished from the SIO level of the keyboard. To get to the SIO level from the TOP level, send an SIO (value = 1) command. The SIO level supports 3 commands. These are:

1) READ operation - command = 0

2) WRITE operation - command = 1

3) RETURN - command = 7 (Returns to top level)

The keyboard SIO support was designed so that the interfaces for READ and WRITE operations were similar. There is plenty of room for common code (see the PIC directory SIO.S). The operations will take care of all time critical code with 2 notable exceptions. One, when reading data from a peripheral, the base unit must read it quickly enough so that the buffer in the keyboard does not overflow. In practice, when reading a disk, this means that DMA can stay on but any interrupt driven code must return quickly. Two, when writing data to a peripheral, the base unit can not spend large amounts of time between sending out data bytes. Again, this means that interrupt driven code must return quickly.

SIO READ OPERATIONS

To reach the SIO READ level, the base unit sets the keyboard to the SIO level and then sends a 0 command. Note that at the conclusion of the SIO READ, the keyboard is back at the SIO level. The keyboard expects the following actions from the base unit code

1) Send the number of bytes to be read.

2) Send the command frame as follows

    a) Send the SIO device ID.

    b) Send the SIO command byte.

    c) Send the 1st auxiliary byte.

    d) Send the 2nd auxiliary byte.

    e) Send the checksum of the command frame (Same checksum as SIO)

PROGRAM_RECORDER_OPERATIONS

The 7800 Keyboard supports the ATARI 1010 Program Recorder. The operations supported are READ record and WRITE record. The record and data formats are identical to those used by Atari Home Computers. Here is a quick summary of those formats. For more detailed information, see the ATARI OS User's Guide and the ATARI Home Computer Hardware Manual.

Data Format

Bits are written at a rate of 600 Baud.

Bits consist of marks (1's) and spaces (0's).

A mark is a frequency of 5327 Hz.

A space is a frequency of 3995 Hz.

Each byte has a start bit which is a space, and a stop bit which is a mark.

There are 132 bytes in a record. The bytes are:

Bytes 1,2) Marker bytes. Each equal to $55. Used by the keyboard to adjust to the actual speed of the program recorder.

Byte 3) Control byte. Gives information data bytes which follow. $FC - full data record (128 bytes). $FA - partial data record, length of data will be in the 128th data byte. $FE - end of file, followed by 128 zero bytes.

Bytes 4-131) Data bytes. If control = $FA, then byte 131 = length of valid data in record.

Byte 132) Checksum of data bytes. Equal to 1 byte sum of all other bytes in record (including marker and control bytes) with the carry added back in each addition and at end of summation.

A file consists of :

1) A 20 second mark tone leader.
2) Any number of data records.
3) An end-of-file record.

Getting to Program Recorder Level

To get to the Program Recorder Level, assuming the keyboard is at the TOP level, send a PR (2) command to the keyboard. Once at the PR level, there are [illegible] commands: ReadRecord, 0; WriteRecord, 1; and RETURN, 7. ReadRecord will read in a data record and remain at the PR level. WriteRecord will write [illegible] data record and remain at the PR level. RETURN will return the keyboard to the TOP level.

READ RECORD

To begin reading in a record from the Program Recorder, send a ReadRecord command (value = 0) to the keyboard. The keyboard will send the following data to the keyboard.

The control value for the record.

128 data bytes.

The checksum for the record. (Note, this includes the two marker bytes which are never sent to the base unit. Init check sum to $AA to account for this.)

A return code byte. If not equal to 0, the error means the program recorder timed out.

Notes:

1) Be sure to have the joystick port properly configured for reading in data.

2) If a bad checksum is received, and its not just bad tape, 1 problem might be that the base unit does too much processing between data bytes and that the keyboard buffer overflows. Too test, try running with DMA off, and just putting data into a buffer.

3) The program recorder will not start until the read record command is issued.

Writing a Record

To get to the WriteRecord level from the PP level, send a WriteRecord command (value = 1) to the keyboard. The keyboard will then write out a data record containing 128 data bytes and remain at the PR level. The keyboard expects the following behavior from the base unit.

1) Send a byte equal to the length of leader tone to be written. The time in seconds is approximately equal to 0.1 * the byte. The first record in a file should be preceded by 20 seconds of leader tone. The standard value for 20 seconds is 208. To maintain compatibility with Atari Home Computers, the pre-record tone (i.e, the leader duration before all other records) should be equal to 3 seconds. The value for 3 seconds is 32. (Note - Atari BASIC and others support a CSAVE which saves tokenized BASIC with leaders of .25 seconds. BASIC for the 7800 does not support this feature and attempts to read a record so written will fail. If it is desired, a value of 1 will write a record with a leader of 0.02 seconds. A data record takes about 0.75 seconds to write, excluding the leader tone.)

2) Send the 2 marker bytes. Value = $55 and $55.

3) Send the control byte.

4) Send 128 data bytes.

5) Send the checksum for all bytes sent, excluding the delay value.

CONFIDENTIAL

This document contains confidential, proprietary information of the GENERAL COMPUTER COMPANY (GENERAL) which may not be copied, disclosed or used except as expressly authorized in writing by GENEFAL.

Technical Notes

1) Problems can arise if a file is written out with short pre-record tones, and then read back assuming long tones. The usual result is a time out error, though a checksum error is sometimes hit.

2) The program recorder has a motor control line. In order to stop the motor, cause the keyboard to exit from the PP level. Thus, a program can read a record, process the data at its leisure, and then read another record; all by stopping the motor. Note that this will not work unless the pre-record tone is 3 seconds long or longer (head skip).

3) Remember that the tape must be positioned manually. As with the SIO hardware, a good check on the 1010 Program Recorder is too make sure it works on an Atari Home Computer.

4) A common checksum bug is to forget the 2 marker bytes.

CASSETTE_RECORDER_OPERATIONS

The 7800 keyboard allows ordinary cassette recorders to be used as storage devices. To use a cassette recorder requires two audio jacks. Use the first to connect the earphone output of a cassette to the left hand connector on the back of the keyboard; the second, to connect the microphone input on a cassette to the right hand connector on the keyboard. In this case, right and left refer to direction when looking at the back of the keyboard. Note that the keyboard has no control over the motor of the cassette.

To reach the CASSETTE level of the keyboard, send a CASSETTE command (value = 3) when the keyboard is at the TOP level. The CASSETTE level has three valid commands: READ record, value = 0; WRITE record, value = 1, and RETURN, value = 7. READ reads a 128 byte data record and remains at the CASSETTE level. WRITE writes a 128 byte data record and remains at the CASSETTE level. RETURN returns the keyboard to the TOP level.

Data Format

The keyboard stores data onto a tape cassette in a manner very similar to that used by the Apple II. Indeed, if a tape player does not work on an Apple, it is unlikely to on this system. Here is a short summary of the data format:

The data is self clocking.

Each bit consists of 2 edges. (An edge is a transition from +5V to ground or vice-versa.)

A one is two edges in 1 milli-second (1000 Hz.)

A zero is two edges in 0.5 milli-seconds (2000 Hz.)

A byte is 8 bits. No start or stop bits.

Each record begins with 2 synchronization edges at 2500 Hz. (a short 0).

A record is 130 bytes. The structure is the same as for the program recorder, except the two marker bytes are omitted. In short:

1) A control byte. $FC - full 128 byte data record. $FA - partial data record, actual length in 128th data byte. $FE - end of file, data all zeroes.

2) 128 data bytes.

3) Checksum of all other bytes in record. Checksum = 1 byte sum of all bytes with carry added back in.

A file consists of 20 seconds of leader tone (all 1's), followed by any number of data records (each with there own leader tone), followed by and end of file record.

READING A RECORD

To set to the READ record level from the CASSETTE level, issue a READ command (value = 0) to the keyboard. The keyboard will transmit the 130 bytes from the next encountered data record and then remain at the CASSETTE level. Remember that the motor can not be controlled by the keyboard. Once the READ is issued, the keyboard will send the following bytes.

The control byte.

128 data bytes.

The checksum byte.

A 1 byte error code. If not = 0, then error is a timeout error.

Notes:

1) If a cassette has a low pass/ high pass filter, proper operation requires that the low pass filter be disabled.

2) The duration of the leader tones before records is under the control of the base unit software. If the leader is too short, processing between records may not be possible.

WRITING A RECORD

To WRITE a record to a cassette tape, issue the WRITE record command (value = 1) from the CASSETTE level. This will write a 130 byte data record to the cassette and then remain at the CASSETTE level. After sending the WRITE command, the base unit code should:

Send the leader tone duration to the keyboard. This 1 byte value causes a leader tone to be written with a value = 0.17 * itself (in seconds). The leader tone before the 1st record of a file should be 20 seconds long, so a value of 117 is required. If the base unit code just reads the other records into a buffer, a value of 1 is appropriate. If, like BASIC, substantial computation is done between records, 3 seconds might be used (value = 18). Remember, the cassette's motor is not under program or keyboard control.

Send the record control byte.

3) Send the 128 data bytes.

4) Send the checksum of the control and data bytes.

TECHNICAL NOTES

1) As with the 1010, files written out with short pre-record leaders, must be read back in in a manner compatible with short leaders.

2) Not all cassettes will work with the keyboard. In general, if it works on an Apple, it will work with a 7800.

7) Be sure to check for timeout errors after a bad checksum. If the keyboard times out, it still sends dummy data. Then, it sends the return code.

THE PIC DIRECTORY

INTRODUCTION

The PIC directory is a directory of 4 source files containing 6502 source code which exercises the 7800 Pro-System keyboard's capabilities. It is called the PIC directory after the PIC 1670 microprocessor in the keyboard. The directory is intended to speed program development for the keyboard. A copy of the source is appended to this manual.

Communication Routines - COMM.S

Every keyboard program is going to need to synchronize to and communicate with the keyboard. This directory contains the routines, and the RAM usage this requires. The key entry points are:

SYNCH () - This routine synchronizes the base unit with the keyboard. It must be called before any other keyboard routine.

SENDB (A) - Send the byte in register A to the keyboard.

READB () - Read a byte from the keyboard. Returned in register A.

SEND3 (A) - Send the 3 least significant bits in A to the keyboard.

READ3 () - Read 3 bits from the keyboard. Returned as 3 LSB's of A.

Notes:

1) All the other source files assume the existence of COMM.S

2) Note that the joystick port is always configured for output except when (while) reading data from the keyboard.

Keyboard Polling - KEY.S

This source file implements keyboard polling. It has a type ahead buffer and handles keycode to ATASCII translation. Key entry point are:

POLL () - Polls the keyboard. Does translation to ATASCII. Handles repeat keys. To work optimally, should be called once each frame (DLI driven?). Note that this POLL routine ignores all Control Shift keys.

PUTC (A) - Puts the contents of A into the type ahead buffer. Used by POLL. Can also be used to insert chars into buffer from main line code.

GETC () - Gets a key from type ahead buffer. Returned in A.

SIO Operations - SIO.S

This source file implements an SIO interface similar to the SIO entry point in the Operating System for the Atari Home Computers. The entry point

for the code is SIO. The code will cause the operation specified by the Device Control Block (label RAMSIO) to be executed. The Y register has the return code as well as SIOSTAT. Note that a good return code is 0, not 1.

```
RAMSIO
SIODEV    DS   1          ; DEVICE ID
SIOCMD    DS   1          ; SIO COMMAND
SIOAUX1   DS   1          ; AUXILIARY BYTE 1
SIOAUX2   DS   1          ; AUXILIARY BYTE 2
SIOADDR   DS   2          ; ADDRESS OF DATA BUFFER
SIOBLEN   DS   1          ; LENGTH OF DATA BUFFER
SIOTIME   DS   1          ; TIME OUT VALUE IN SECONDS
SIOSTAT   DS   1          ; RETURN CODE
```

Notes :

1) If the SIO command is a write operation, the upper bit SIOCMD should be turned on. Thus, PUT SECTOR (WITH VERIFY) is not $57, but $D7.

2) If this code does not work in your program, try running it without DMA. If that fixes the problem, the likely explanation is that your interrupt driven code runs too long; short circuit it with a flag. If it still bombs with DMA off, make sure that the keyboard is synched before SIO executes and that the communication routines work.

3) Another common one is interrupt driven code attempting to poll the keyboard whilst the SIO code is trying to execute.

Program Recorder/ Cassette Operation - TAPE.S

This source file implements read and write record for both the 1010 program recorder and cassette recorders. There is a high degree of common code between the 2. The key entry points are:

READREC () - Reads a data record into TAPEBUFF. Sets up TAPELEN and TAPECTRL

WRITEREC (A) - Writes a data record from the data in TAPEBUFF. Relies on TAPECTRL and TAPELEN.

Note - the LSB of TAPEFLAG is set for cassettes and clear for the 1010. Bit 6 of the same flag is used by WRITEREC to determine if the record is the 1st to be written. This file is somewhat wasteful of RAM.

FINAL WORDS

This is the preliminary version of the software guide. As such, it probably contains some (hopefully few) errors. Any errors should be brought to my attention so I can revise the guide. Suggestions for revising the format are also encouraged. Good luck.

6/19/84 2:35 am